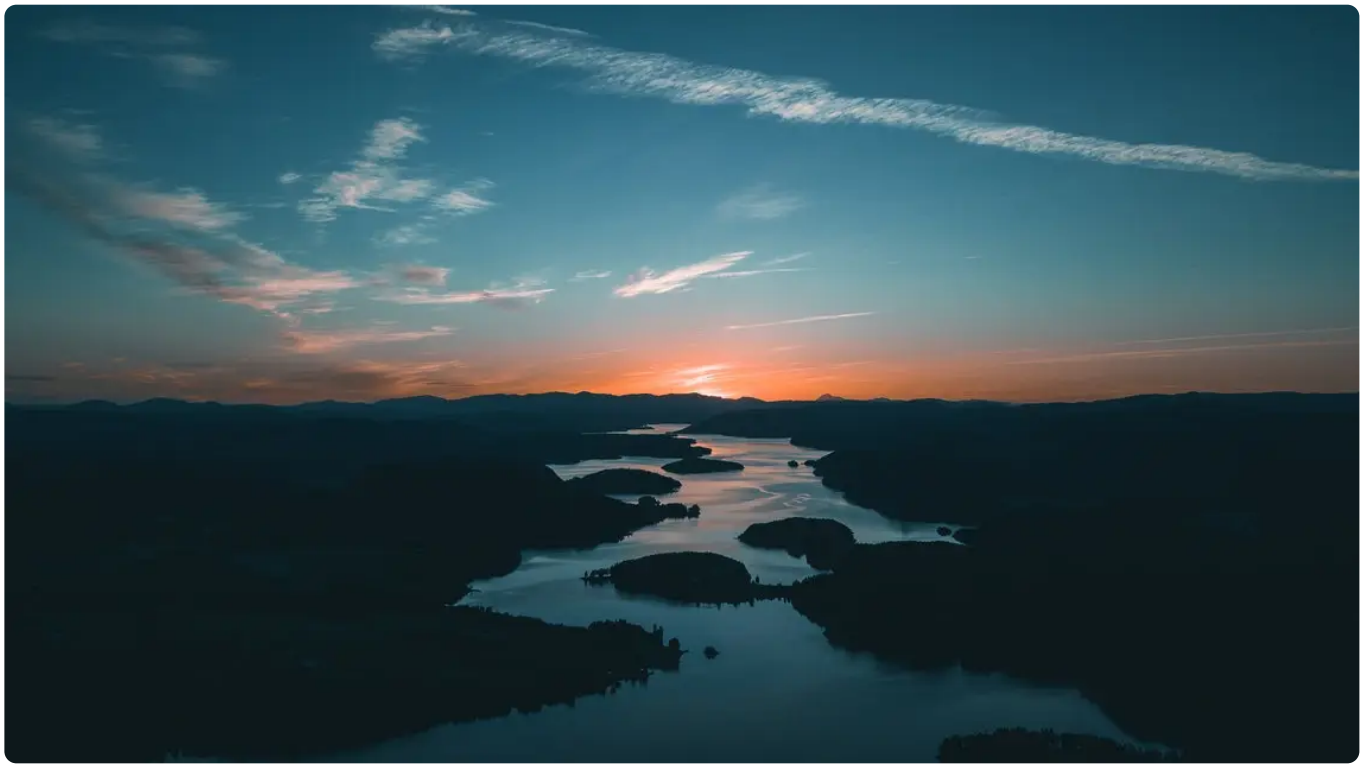
04 | 运行原理：eBPF 是一个新的虚拟机吗？

2022-01-24 倪朋飞

《eBPF 核心技术与实战》 课程介绍 >



讲述：山荣
时长 13:21 大小 12.24M

你好，我是倪朋飞。

上一讲，我带你一起搭建了 eBPF 的开发环境，并从最简单的 Hello World 开始，带你借助 BCC 库从零开发了一个跟踪 openat() 系统调用的 eBPF 程序。

不过，虽然第一个 eBPF 程序已经成功运行起来了，你很可能还在想：这个 eBPF 程序到底是如何编译成内核可识别的格式的？又是如何在内核中运行起来的？还有，既然允许普通用户去修改内核的行为，它又是如何确保内核安全的呢？

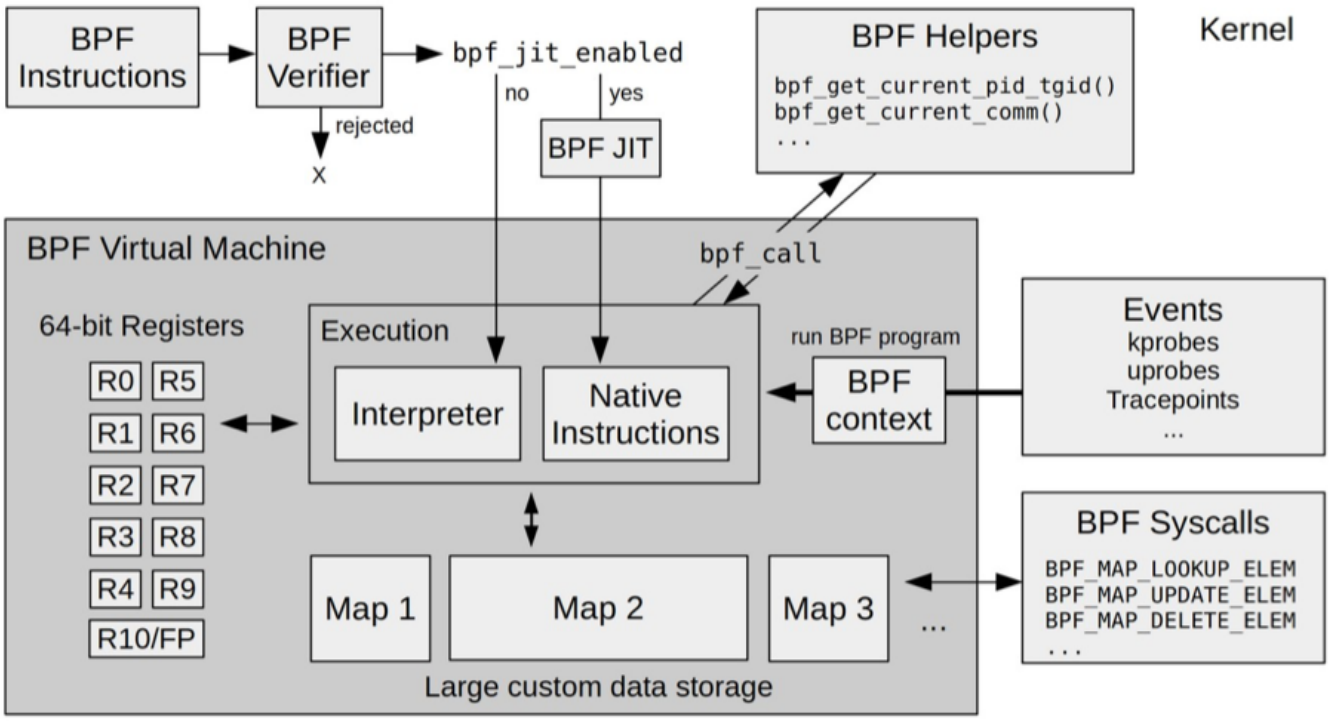
今天，我就带你一起深入看看 eBPF 虚拟机的原理，以及 eBPF 程序是如何执行的。

eBPF 虚拟机是如何工作的？

eBPF 是一个运行在内核中的虚拟机，很多人在初次接触它时，会把它跟系统虚拟化（比如 kvm）中的虚拟机弄混。其实，虽然都被称为“虚拟机”，系统虚拟化和 eBPF 虚拟机还是有着本质不同的。

系统虚拟化基于 x86 或 arm64 等通用指令集，这些指令集足以完成完整计算机的所有功能。而为了确保在内核中安全地执行，eBPF 只提供了非常有限的指令集。这些指令集可用于完成一部分内核的功能，但却远不足以模拟完整的计算机。为了更高效地与内核进行交互，eBPF 指令还有意采用了 C 调用约定，其提供的辅助函数可以在 C 语言中直接调用，极大地方便了 eBPF 程序的开发。

如下图（图片来自 BPF Internals）所示，eBPF 在内核中的运行时主要由 5 个模块组成：



eBPF 运行时

第一个模块是 **eBPF 辅助函数**。它提供了一系列用于 eBPF 程序与内核其他模块进行交互的函数。这些函数并不是任意一个 eBPF 程序都可以调用的，具体可用的函数集由 BPF 程序类型决定。关于 BPF 程序类型，我会在 06 讲 中进行讲解。

第二个模块是 **eBPF 验证器**。它用于确保 eBPF 程序的安全。验证器会将待执行的指令创建为一个有向无环图（DAG），确保程序中不包含不可达指令；接着再模拟指令的执行过程，确保不会执行无效指令。

第三个模块是由 **11 个 64 位寄存器、一个程序计数器和一个 512 字节的栈组成的存储模块**。这个模块用于控制 eBPF 程序的执行。其中，R0 寄存器用于存储函数调用和 eBPF 程序的返回值，这意味着函数调用最多只能有一个返回值；R1-R5 寄存器用于函数调用的参数，因此函数调用的参数最多不能超过 5 个；而 R10 则是一个只读寄存器，用于从栈中读取数据。

第四个模块是**即时编译器**，它将 eBPF 字节码编译成本地机器指令，以便更高效地在内核中执行。

第五个模块是 **BPF 映射（map）**，它用于提供大块的存储。这些存储可被用户空间程序用来进行访问，进而控制 eBPF 程序的运行状态。

关于 BPF 辅助函数和 BPF 映射的具体内容，我在后面的课程中还会为你详细介绍。接下来，我们先来看看 BPF 指令的具体格式，以及它是如何加载到内核中，又是何时运行的。

BPF 指令是什么样的？

只看图中的这些模块，你可能觉得它们并不是太直观。所以接下来，我们还是用上一讲的 Hello World 作为例子，一起看下 BPF 指令到底是什么样子的。

首先，回顾一下上一讲的 eBPF 程序 Hello World 的源代码。它的逻辑其实很简单，先调用 `bpf_trace_printk` 输出一个 “Hello, World!” 字符串，然后就返回成功了：

复制代码

```
int hello_world(void *ctx)
{
  bpf_trace_printk("Hello, World!");
  return 0;
}
```

然后，我们通过 BCC 的 Python 库，加载并运行了这个 eBPF 程序：

复制代码

```
#!/usr/bin/env python3
# This is a Hello World example of BPF.
from bcc import BPF

# load BPF program
b = BPF(src_file="hello.c")
```

```
7 b.attach_kprobe(event="do_sys_openat2", fn_name="hello_world")
8 b.trace_print()
```

在终端中运行下面的命令，就可以启动这个 eBPF 程序（注意， BCC 帮你完成了编译和加载的过程）：

复制代码

```
1 sudo python3 hello.py
```

接下来，我为你介绍一个新的工具 bpftool，用它可以查看 eBPF 程序的运行状态。

首先，打开一个新的终端，执行下面的命令，查询系统中正在运行的 eBPF 程序：

复制代码

```
1 # sudo bpftool prog list
2 89: kprobe  name hello_world  tag 38dd440716c4900f  gpl
3       loaded_at 2021-11-27T13:20:45+0000  uid 0
4       xlated 104B  jited 70B  memlock 4096B
5       btf_id 131
6       pids python3(152027)
```

输出中，89 是这个 eBPF 程序的编号，kprobe 是程序的类型，而 hello_world 是程序的名字。

有了 eBPF 程序编号之后，执行下面的命令就可以导出这个 eBPF 程序的指令（注意把 89 替换成你查询到的编号）：

复制代码

```
1 sudo bpftool prog dump xlated id 89
```

你会看到如下所示的输出：

复制代码

```
1 int hello_world(void * ctx):
2 ; int hello_world(void *ctx)
```

```
3     0: (b7) r1 = 33                   /* ! */
4  ; ({ char _fmt[] = "Hello, World!"; bpf_trace_printk_(_fmt, sizeof(_fmt)); });
5     1: (6b) *(u16 *)(r10 -4) = r1
6     2: (b7) r1 = 1684828783           /* dlro */
7     3: (63) *(u32 *)(r10 -8) = r1
8     4: (18) r1 = 0x57202c6f6c6c6548   /* W ,olleH */
9     6: (7b) *(u64 *)(r10 -16) = r1
10    7: (bf) r1 = r10
11 ;
12    8: (07) r1 += -16
13 ; ({ char _fmt[] = "Hello, World!"; bpf_trace_printk_(_fmt, sizeof(_fmt)); });
14    9: (b7) r2 = 14
15   10: (85) call bpf_trace_printk#-61616
16 ; return 0;
17   11: (b7) r0 = 0
18   12: (95) exit
```

其中，分号开头的部分，正是我们前面写的 C 代码，而其他行则是具体的 BPF 指令。具体每一行的 BPF 指令又分为三部分：

第一部分，冒号前面的数字 0-12 ，代表 BPF 指令行数；

第二部分，括号中的 16 进制数值，表示 BPF 指令码。它的具体含义你可以参考 IOVisor BPF 文档，比如第 0 行的 0xb7 表示为 64 位寄存器赋值。

第三部分，括号后面的部分，就是 BPF 指令的伪代码。

结合前面讲述的各个寄存器的作用，不难理解这些 BPF 指令的含义：

第 0-8 行，借助 R10 寄存器从栈中把字符串 “Hello, World!” 读出来，并放入 R1 寄存器中；

第 9 行，向 R2 寄存器写入字符串的长度 14（即代码注释里面的 `sizeof(_fmt)`）；

第 10 行，调用 BPF 辅助函数 `bpf_trace_printk` 输出字符串；

第 11 行，向 R0 寄存器写入 0，表示程序的返回值是 0；

最后一行，程序执行成功退出。

总结起来，**这些指令先通过 R1 和 R2 寄存器设置了 `bpf_trace_printk` 的参数，然后调用 `bpf_trace_printk` 函数输出字符串，最后再通过 R0 寄存器返回成功。**

实际上，你也可以通过类似的 BPF 指令来开发 eBPF 程序（具体指令的定义，请参考 include/uapi/linux/bpf_common.h 以及 include/uapi/linux/bpf.h），不过通常并不推荐你这么做。跟一开始的 C 程序相比，你会发现 BPF 指令的可读性和可维护性明显要差得多。所以，我建议你还是使用 C 语言来开发 eBPF 程序，而只把 BPF 指令作为排查 eBPF 程序疑难杂症时的参考。

这里，我来简单讲讲 BPF 指令加载后是如何运行的。当这些 BPF 指令加载到内核后，BPF 即时编译器会将其编译成本地机器指令，最后才会执行编译后的机器指令：

复制代码

```
# bpftool prog dump jited id 89
int hello_world(void * ctx):
bpf_prog_38dd440716c4900f_hello_world:
; int hello_world(void *ctx)
   0:  nopl   0x0(%rax,%rax,1)
   5:  xchg   %ax,%ax
   7:  push   %rbp
   8:  mov    %rsp,%rbp
   b:  sub    $0x10,%rsp
  12:  mov    $0x21,%edi
; ({ char _fmt[] = "Hello, World!"; bpf_trace_printk_(_fmt, sizeof(_fmt)); });
  17:  mov    %di,-0x4(%rbp)
  1b:  mov    $0x646c726f,%edi
  20:  mov    %edi,-0x8(%rbp)
  23:  movabs $0x57202c6f6c6c6548,%rdi
  2d:  mov    %rdi,-0x10(%rbp)
  31:  mov    %rbp,%rdi
;
  34:  add    $0xfffffffffffffff0,%rdi
; ({ char _fmt[] = "Hello, World!"; bpf_trace_printk_(_fmt, sizeof(_fmt)); });
  38:  mov    $0xe,%esi
  3d:  call   0xffffffffd8c7e834
; return 0;
  42:  xor    %eax,%eax
  44:  leave
  45:  ret
```

这些机器指令的含义跟前面的 BPF 指令是类似的，但具体的指令和寄存器都换成了 x86 的格式。你不需要掌握这些机器指令的具体含义，只要知道查询的具体方法就足够了。这是因为，就像你曾接触过的其他高级语言一样，在实际的 eBPF 使用过程中，并不需要直接使用机器指令，而是 eBPF 虚拟机帮你自动完成了转换。

eBPF 程序是什么时候执行的？

到这里，我想你已经理解了 BPF 指令的具体格式，以及它与 C 源代码之间的对应关系。不过，这个 eBPF 程序到底是什么时候执行的呢？接下来，我们再一起看看 BPF 指令的加载和执行过程。

在上一讲中我提到，BCC 负责了 eBPF 程序的编译和加载过程。因而，要了解 BPF 指令的加载过程，就可以从 BCC 执行 eBPF 程序的过程入手。

那么，怎么才能查看到 BCC 的执行过程呢？我想，你一定想到了，那就是跟踪它的系统调用过程。

首先，我们打开一个终端，执行下面的命令：

复制代码

```
# -ebpf表示只跟踪bpf系统调用
sudo strace -v -f -ebpf ./hello.py
```

稍等一会，你会看到如下的输出：

复制代码

```
bpf(BPF_PROG_LOAD,
    {
        prog_type=BPF_PROG_TYPE_KPROBE,
        insn_cnt=13,
        insns=[
            {code=BPF_ALU64|BPF_K|BPF_MOV, dst_reg=BPF_REG_1, src_reg=BPF_REG_
            {code=BPF_STX|BPF_H|BPF_MEM, dst_reg=BPF_REG_10, src_reg=BPF_REG_1
            {code=BPF_ALU64|BPF_K|BPF_MOV, dst_reg=BPF_REG_1, src_reg=BPF_REG_
            {code=BPF_STX|BPF_W|BPF_MEM, dst_reg=BPF_REG_10, src_reg=BPF_REG_1
            {code=BPF_LD|BPF_DW|BPF_IMM, dst_reg=BPF_REG_1, src_reg=BPF_REG_0,
            {code=BPF_LD|BPF_W|BPF_IMM, dst_reg=BPF_REG_0, src_reg=BPF_REG_0,
            {code=BPF_STX|BPF_DW|BPF_MEM, dst_reg=BPF_REG_10, src_reg=BPF_REG_
            {code=BPF_ALU64|BPF_X|BPF_MOV, dst_reg=BPF_REG_1, src_reg=BPF_REG_
            {code=BPF_ALU64|BPF_K|BPF_ADD, dst_reg=BPF_REG_1, src_reg=BPF_REG_
            {code=BPF_ALU64|BPF_K|BPF_MOV, dst_reg=BPF_REG_2, src_reg=BPF_REG_
            {code=BPF_JMP|BPF_K|BPF_CALL, dst_reg=BPF_REG_0, src_reg=BPF_REG_0
            {code=BPF_ALU64|BPF_K|BPF_MOV, dst_reg=BPF_REG_0, src_reg=BPF_REG_
            {code=BPF_JMP|BPF_K|BPF_EXIT, dst_reg=BPF_REG_0, src_reg=BPF_REG_0
        ],
        prog_name="hello_world",
        ...
```

```
22      },
23      128) = 4
```

这些参数看起来很复杂，但实际上，如果你查询 bpf 系统调用的格式（执行 man bpf 命令），就可以发现，它实际上只需要三个参数：

复制代码

```
1 int bpf(int cmd, union bpf_attr *attr, unsigned int size);
```

对应前面的 strace 输出结果，这三个参数的具体含义如下。

第一个参数是 BPF_PROG_LOAD ，表示加载 BPF 程序。

第二个参数是 bpf_attr 类型的结构体，表示 BPF 程序的属性。其中，有几个需要你留意的参数，比如：

- prog_type 表示 BPF 程序的类型，这儿是 BPF_PROG_TYPE_KPROBE ，跟我们 Python 代码中的 attach_kprobe 一致；
- insn_cnt (instructions count) 表示指令条数；
- insns (instructions) 包含了具体的每一条指令，这儿的 13 条指令跟我们前面 bpftool prog dump 的结果是一致的（具体的指令格式，你可以参考内核中 bpf_insn 的定义）；
- prog_name 则表示 BPF 程序的名字，即 hello_world 。

第三个参数 128 表示属性的大小。

到这里，我们已经了解了 bpf 系统调用的基本格式。对于 bpf 系统调用在内核中的实现原理，你并不需要详细了解。我们只要知道它的具体功能，就可以掌握 eBPF 的核心原理了。当然，如果你对它的实现方法有兴趣的话，可以参考内核源码 kernel/bpf/syscall.c 中 SYSCALL_DEFINE3 的实现。

BPF 程序加载到内核后，并不会立刻执行，那么它什么时候才会执行呢？这里，回想一下我在 01 讲 中提到的 eBPF 的基本原理：

> eBPF 程序并不像常规的线程那样，启动后就一直运行在那里，它需要事件触发后才会执行。这些事件包括系统调用、内核跟踪点、内核函数和用户态函数的调用退出、网络事件，等等。

对于我们的 Hello World 来说，由于调用了 `attach_kprobe` 函数，很明显，这是一个内核跟踪事件：

复制代码

```
b.attach_kprobe(event="do_sys_openat2", fn_name="hello_world")
```

所以，除了把 eBPF 程序加载到内核之外，还需要把加载后的程序跟具体的内核函数调用事件进行绑定。在 eBPF 的实现中，诸如内核跟踪（kprobe）、用户跟踪（uprobe）等的事件绑定，都是通过 `perf_event_open()` 来完成的。

为什么这么说呢？我们再用 `strace` 来确认一下。把前面 `strace` 命令中的 `-ebpf` 参数去掉，重新执行：

复制代码

```
sudo strace -v -f ./hello.py
```

忽略无关的输出后，你会发现如下的系统调用：

复制代码

```
...
/* 1) 加载BPF程序 */
bpf(BPF_PROG_LOAD,...) = 4
...

/* 2) 查询事件类型 */
openat(AT_FDCWD, "/sys/bus/event_source/devices/kprobe/type", O_RDONLY) = 5
read(5, "6\n", 4096)                    = 2
close(5)                                = 0
...

/* 3) 创建性能监控事件 */
perf_event_open(
    {
        type=0x6 /* PERF_TYPE_??? */,
```

```
        size=PERF_ATTR_SIZE_VER7,
        ...
        wakeup_events=1,
        config1=0x7f275d195c50,
        ...
    },
    -1,
    0,
    -1,
    PERF_FLAG_FD_CLOEXEC) = 5

/* 4 ) 绑定BPF到kprobe事件 */
ioctl(5, PERF_EVENT_IOC_SET_BPF, 4)     = 0

```

从输出中，你可以看出 BPF 与性能事件的绑定过程分为以下几步：

首先，借助 bpf 系统调用，加载 BPF 程序，并记住返回的文件描述符；

然后，查询 kprobe 类型的事件编号。BCC 实际上是通过 `/sys/bus/event_source/devices/kprobe/type` 来查询的；

接着，调用 `perf_event_open` 创建性能监控事件。比如，事件类型（type 是上一步查询到的 6）、事件的参数（ `config1` 包含了内核函数 `do_sys_openat2` ）等；

最后，再通过 `ioctl` 的 `PERF_EVENT_IOC_SET_BPF` 命令，将 BPF 程序绑定到性能监控事件。

对于绑定性能监控（perf event）的内核实现原理，你也不需要详细了解，只需要知道它的具体功能，就足够我们掌握 eBPF 了。如果你对它的实现方法有兴趣的话，可以参考内核源码 perf_event_set_bpf_prog 的实现；而最终性能监控调用 BPF 程序的实现，则可以参考内核源码 kprobe_perf_func 的实现。

小结

今天，我带你一起梳理了 eBPF 在内核中的实现原理，并以上一讲的 Hello World 程序为例，借助 bpftool、strace 等工具，带你观察了 BPF 指令的具体格式。

然后，我们从 BCC 执行 eBPF 程序的过程入手，一起看了 BPF 指令的加载和执行过程。用高级语言开发的 eBPF 程序，需要首先编译为 BPF 字节码（即 BPF 指令），然后借助

bpf 系统调用加载到内核中，最后再通过性能监控等接口，与具体的内核事件进行绑定。这样，内核的性能监控模块才会在内核事件发生时，自动执行我们开发的 eBPF 程序。

思考题

最后，我想邀请你来聊一聊这两个问题。

1. 你通常是如何快速理解一门新技术的运行原理的？
2. 在今天的内容中，我使用 strace 跟踪 BCC 程序，进而找到了相关的系统调用。那么，有没有可能直接使用 BCC 来跟踪 bpf 系统调用呢？如果你的答案是肯定的，可以试着把它开发出来，并在评论区分享你的实践经验。

欢迎在留言区和我讨论，也欢迎把这节课分享给你的同事、朋友。我们一起在实战中演练，在交流中进步。

分享给需要的人，Ta订阅超级会员，你将得 50 元

Ta单独购买本课程，你将得 20 元

生成海报并分享

赞 8 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 03 | 初窥门径：开发并运行你的第一个 eBPF 程序

下一篇 05 | 编程接口：eBPF 程序是怎么跟内核进行交互的？

精选留言 (8)

写留言

莫名

2022-01-24

追踪 bpf 系统调用，借助 BCC 宏定义 TRACEPOINT_PROBE(category, event) 比较方便，例如：

---------- example.c ----------

...

展开

作者回复: 谢谢分享实践经验👍，BCC内置的很多宏的确非常方便。

共 2 条评论 10

18646333118

2022-02-09

解决方法:

{

"features": {

"libbfd": false

}...

展开

作者回复: 👍 谢谢分享源码编译bpftool的详细步骤！

七里

2022-02-02

请问，不能执行'bpftool prog dump jited id 78'是怎么回事？bpf相关的包都按照上一讲的提示按照上了

root@maqi-ubt:~# bpftool prog dump xlated id 78

int hello_world(void * ctx):...

展开

作者回复: 可以运行下面的命令来查询bpftool支持的特性：

sudo bpftool version -p

如果出现下面的输出说明发行版自带的bpftool默认不支持libbfd，需要下载内核源码并安装binutils-dev之后重新编译bpftool：

{

"features": {

"libbfd": false

```
    }
}
```

写点啥呢
2022-01-29

请问老师，像上节课例子中 trace openat系统调用的这个函数int hello_world(struct pt_regs *ctx, int dfd, const char __user * filename, struct open_how *how)，bpf会自动把系统调用参数注入bpf函数执行中。本节课提到bpf虚拟机中对bpf函数的参数个数有限制，那如果碰到系统调用参数个数大于bpf限制了，该如何处理呢？
...
展开

作者回复: 系统调用最多只有6个参数，不会更多的，具体可以参考内核源码 https://elixir.bootlin.com/linux/v5.13/source/include/linux/syscalls.h#L216。

共 2 条评论

不了峰
2022-01-28

你通常是如何快速理解一门新技术的运行原理的？
--- 看一下官方文档，了解体系架构，多看几遍。买书看感觉也是一个快速入门的方法。但是对于没有编程经验，对于 字节码、cpu寄存器、jit ，编译器的理解还是很抽象，学到这里还是有点晕。感觉还是要把这课程从头再看一遍。
展开

作者回复: 谢谢分享。我的理解是，要深入一门技术的每个细节需要看很多书籍，但一开始要有个侧重点，不要发散的太广了。

不了峰
2022-01-27

root@ubuntu-impish:~# bpftool prog dump jited id 331
Error: No libbfd support
root@ubuntu-impish:~#
---
root@ubuntu-impish:~# bpftool -V...

展开

作者回复: 可以运行下面的命令来查询bpftool支持的特性：

sudo bpftool version -p

如果出现下面的输出说明默认不支持libbfd，需要下载内核源码并安装binutils-dev之后重新编译bpftool：

```
{
    "features": {
        "libbfd": false
    }
}
```

ZR2021

2022-01-27

老师，我这边执行 bpftool prog dump jited id xxx，报"Error: No libbfd support" 错误，网上也没找到原因，系统是最新的ubuntu 21.10，vmvare的虚拟机，执行xlated是可以打印出指令的，不知道啥个情况

作者回复: 可以运行下面的命令来查询bpftool支持的特性：

sudo bpftool version -p

如果出现下面的输出说明默认不支持libbfd，需要下载内核源码并安装binutils-dev之后重新编译bpftool：

```
{
    "features": {
        "libbfd": false
    }
}
```


共 3 条评论 >

ZR2021

2022-01-26

老师，sudo bpftool prog dump xlated id 89，这条命令输出的指令是存储模块存储的指令吗，这个指令是在验证器里进行转换的吗，验证器用转换后的指令去模拟执行是否安全，

对bpf 接触的比较少，可能问的有点低级.....

作者回复: 验证器是发生在加载过程中，而这儿看到的是已经加载到内核内存中了（验证失败就不会加载了）

共 2 条评论 >